Declarative Experimentation in Information Retrieval using PyTerrier

Craig Macdonald
University of Glasgow

Nicola Tonellotto
University of Pisa

ABSTRACT

The advent of deep machine learning platforms such as Tensorflow and Pytorch, developed in expressive high-level languages such as Python, have allowed more expressive representations of deep neural network architectures. We argue that such a powerful formalism is missing in information retrieval (IR), and propose a framework called PyTerrier that allows advanced retrieval pipelines to be expressed, and evaluated, in a declarative manner close to their conceptual design. Like the aforementioned frameworks that compile deep learning experiments into primitive GPU operations, our framework targets IR platforms as backends in order to execute and evaluate retrieval pipelines. Further, we can automatically optimise the retrieval pipelines to increase their efficiency to suite a particular IR platform backend. Our experiments, conducted on TREC Robust and ClueWeb09 test collections, demonstrate the efficiency benefits of these optimisations for retrieval pipelines involving both the Anserini and Terrier IR platforms.

ACM Reference Format:
Craig Macdonald and Nicola Tonellotto. 2020. Declarative Experimentation in Information Retrieval using PyTerrier. In *2020 ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR '20), September 14–17, 2020, Virtual Event, Norway*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409256.3409829

1 INTRODUCTION

Information retrieval (IR) is classically an empirical science. Offline experiments towards enhancing retrieval effectiveness have been easily made possible through use of test collections with documents judged for relevance by human assessors, as typified by the TREC, CLEF, NCTIR evaluation forums.

On the other hand, machine learning has experienced even greater growth, with applications to many areas of science, driven by the availability of good datasets [9], as well as platforms that allow easy development and application of machine learned models. In recent years, there has been a focus on the development and application of deep learning frameworks written in high-level languages, including Lua (Torch), but particularly Python (Tensorflow and Pytorch). Using such expressive high-level languages allow complex deep neural network architectures with various matrix operations to be expressed using familiar programming paradigms, for instance, adding matrices using a + operator, or adding several hidden layers using a `for` loop to add objects to a list.

We argue that adoption of such an expressive high-level languages are missing from many of the available IR platforms, and hence we are unable to perform wide-ranging experiments with the ease of our machine learning compatriots. End-to-end retrieval evaluation is not easy to perform. For example, Figure 1 depicts, in the form of a directed acyclic graph (DAG), an example IR experimental workflow to compare the effectiveness of two IR systems involving the setup and execution of different classical IR techniques, including ranked retrieval with different weighting models, fusion, features extraction, learning to rank (LTR) algorithms and neural re-ranking. Typically, conducting such experiments involves editing several configuration files, running various commands to generate result files to be fed to the following stages, and eventually invoking `trec_eval` to evaluate the experimental outcomes. Configuration is spread across several files making reproducibility difficult.

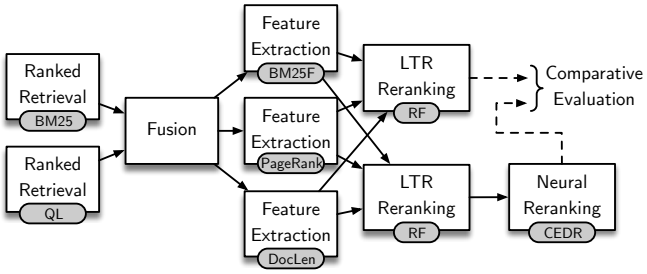

Figure 1: An experiment comparing two complex retrieval pipelines, with learning-to-rank and neural re-ranking.

Yet reproducibility is key to impactful science. Ferro & Kelly [10] define reproducibility as the ability for a different team to reproduce the measurement in a different experimental setup. Therefore, focussing evaluation solely on datasets that extract key aspects of a problem using a standard dataset – for instance, evaluating LTR techniques solely on LETOR datasets [25] with common features – does not allow us to understand the wider context, such as how an approach will fare when integrated into a fully-fledged search engine's retrieval stack. This highlights the importance of end-to-end retrieval experiments – understanding what data are needed for a given approach, and how it interacts with others components (e.g., how many documents should be re-ranked [19] for a LTR approach), reduces the uncertainties when a technique should be deployed to an operational search engine.

Thus, we argue that a succinct manner of describing a retrieval experiment, in a *conceptual* yet familiar way, should allow more IR researchers with increased ability to develop techniques that can be easily integrated, and evaluated, in an end-to-end fashion. Hence, in this paper, we describe a new framework called PyTerrier[1] for expressing IR experiments with composable *pipelines*. Similar to Tensorflow and Pytorch, it uses Python as a high-level language for operationalising of experiments. Moreover, we use standard operators to combine objects representing retrieval building blocks called *transformers*, allowing advanced retrieval pipelines to be specified in a declarative rather than procedural fashion. We have initially instantiated PyTerrier on two existing IR platforms (Anserini and

ICTIR '20, September 14–17, 2020, Virtual Event, Norway

[1] https://github.com/terrier-org/pyterrier

Terrier), and demonstrate that the transformers and operators can be used to retrieve from, and even combine, multiple systems.

Moreover, the expression of a retrieval pipeline in a high-level language using transformers and operators, forms a DAG of basic IR operations (retrieve, re-rank, combine results, rewrite query, etc). This DAG can be rewritten, optimised and/or passed to the underlying search engine. Hence, a complex retrieval architecture trialled in PyTerrier might be compiled down to an optimized configuration of the underlying search engine(s). Indeed, we show how simple optimisations steps on the DAG result in markedly enhanced efficiency when such system-specific optimisations are employed.

Therefore, this paper contributes a conceptual framework for a different way of envisaging and executing information retrieval experiments. We propose a clear semantics for different IR experiment's building blocks and their composition. The proposed framework is easily extensible, since it allows the definition and inclusion of new IR activities. Moreover, we provide an implementation of our proposed framework in Python, supporting the compilation of IR pipelines on top of two widely-used Java-based IR platforms to conduct fast retrieval operations. Finally, we show how these operations can be optimised – while retaining their semantics – by applying *graph rewriting patterns* targetting the underlying IR platform operations. Moreover, it allows easy integration of deep learning techniques into the retrieval pipeline, thereby allowing to enhance the reproducibilty of retrieval technologies.

The remainder of this paper is structured as follows: Section 2 positions our framework with respect to different standard toolkits, for data science, big data, machine learning and information retrieval; In Section 3, we formally describe our framework in terms of transformers, operators, and experiments. Section 4 discusses how pipelines can be compiled and optimised through *graph rewriting* to be efficiently implemented using less calls to the underlying IR platforms. Section 5 demonstrates the empirical efficiency in these pipeline rewrites through experiments using PyTerrier based on two underlying search engines, namely Anserini and Terrier. We provide concluding remarks and detail future improvements in Section 6.

2 RELATED WORK

IR platforms have a long history, dating back to at least SMART [2]. These days, among the open source platforms, Apache Lucene is widely deployed. Implemented in Java, it provides indexing and single-search APIs, and in recent years has adopted BM25 along with LTR [7] and dynamic pruning techniques [11]. However, its ability to handle standard test collections was for many years a known limitation [8], and has been advanced by efforts such as Anserini [29]. Indeed Anserini facilitates the deployment of a number of replicable information retrieval techniques, on standard test collection, on top of the Lucene backend.

Among the other commonly used academic platform are Lemur/Indri (implemented in C) – along the closely related Galago (implemented in Java) [4] – as well as Terrier (implemented in Java) [18] and PISA [21] (implemented in C++). We note that while Python "bindings" exist for various platforms, including Indri, Galago and Anserini, there are no serious contenders for IR platforms written natively in Python. We believe there are several reasons for this, including the challenge of indexing and retrieving from corpora that contains 1-10 million documents, as well as the more recent maturity of Python. Indri and its modern replacement Galago also have rich (domain-specific) query languages that allow the expression of complex retrieval operations.

All of the discussed IR platforms mix the design of experimental retrieval activities with the implementation and optimisations required to make such activities efficient. This approach has been shown to limit the reproducibility of IR experiments. For example, Mühlisen *et al.* [23] show that different implementations of the same BM25 weighting models in different IR platforms result in different values for the same effectiveness metric. They propose to decouple the IR experiments from the IR platform implementation by storing the inverted index in a column-oriented relational database and by implementing ranking models using SQL. Kamphuis & de Vries [12] take a step forward and propose the adoption of an IR-specific declarative language to provide higher level abstractions in the implementation of the IR experiments based on a graph query language. In contrast to this and the Indri/Galago domain-specific query language, we propose a declarative framework to express basic retrieval operations and their composition using queries and documents as inputs and outputs. It is built upon Python, which is expressive, readily accessible and allows integration with other modern Python toolkits such as those for deep learning. Together, this allows for rapid prototyping and improved reproducibility in IR. We then show how the elements of the proposed declarative language can be compiled into a DAG representation, which can be efficiently implemented on specific IR platforms.

More generally, Python is popular among other branches of data science and machine learning – it has no need to compile, and hence allows researchers/developers to easily adjust their code and re-run, in an agile manner, often using notebook environments such as Jupyter or Google Colab; Standard toolkits such as Pandas (for structured data processing in relational dataframes) and scikit-learn (for machine learning) exemplify this approach to data science.

To allow the efficient application of data processing and machine learning at scale, Apache Spark overcame some of the disadvantages of the store-then-compute MapReduce programming paradigm; Apache Spark, which has bindings in Java, Scala and Python, allows structured data processing operations to be vastly parallelised across a cluster of machines [30]. As a functional language, expressions in Apache Spark, even in Python, are compiled into an execution plan, which is then distributed to different compute machines. We are inspired by this notion of an execution plan.

In recent times, Tensorflow [9] and PyTorch have been the dominant frameworks for neural machine learning. Both are based on primitive operations (add, multiple, concatenate) on tensors, which are expressed in Python by overloading the math operators in the language for the tensor objects; higher level tensor operations such as recurrent units, attention etc., can be achieved using higher level objects. Thus a domain-specific programming environment is created in Python. The object graph instantiated by the tensor objects and operators creates a dataflow DAG, which can be compiled into GPU operations for efficient computation.

In this work, we are inspired by these existing deep learning frameworks, as we both instantiate a domain-specific programming environment in Python, but this time suited for IR experiments. Moreover, similar to deep learning, we observe that search pipelines expressed in this manner form DAGs, which can be compiled into more efficient low-level search engine operations.

The most similar work to our own is that in Terrier-Spark [16, 17], where retrieval pipelines for the Terrier platform were created in

Scala using Apache Spark. In that work, retrieval operation were expressed as operations on dataframes (relations). However, adoption of that framework was hindered by two factors: firstly, the use of Apache Spark, which is designed for processing massive scale datasets, and introduces significant interactive overheads making Terrier-Spark unsuitable for notebook-style agile development; secondly, the use of the Scala programming language, which is not as popular as Python. In this paper, we extend the notion of retrieval operations on dataframes, but instead, operate on Python, and create a domain-specific programming environment where complex retrieval pipelines can be formulated as operations on Python objects.

3 DECLARATIVE RETRIEVAL OPERATIONS

In this section we discuss the IR data model we use to represent queries and documents (Sec. 3.1), we introduce IR *transformers* to express IR operation as the basic element of our declarative IR language (Sec. 3.2) and *operators* to compose transformers into more complex IR *pipelines* (Sec. 3.3), and finally we discuss how to design IR experiments with our declarative IR language (Sec. 3.4).

3.1 IR Data Model

The basic elements of an IR systems are *queries* and *documents*. A query q is a textual representation of an information need, while a document d is a textual source of information. From an object relational model perspective, q and d are *tuples*, i.e., finite sequences of attributes, whose domains and values depends on their associated metadata. These metadata can vary, but in the following we will assume a query q encapsulates at least the query text $q.text$ and the query identifier $q.id$, which also constitutes the primary key. Similarly, we will assume a document d encapsulates at least its full text $d.text$, and the document identifier $d.id$, which also represents the primary key. Both queries and documents may have additional attributes, such as the number of unique terms, the document's URL, title and TREC docno, title, and so on. We denote with $Q = [q_1, q_2, \ldots]$ a finite ordered list of queries, and with $D = [d_1, d_2, \ldots]$ a finite ordered list of documents. From an object relational model perspective, both Q and D are *relations*.

Further, to allow evaluation, we formally define relevance assessments (in TREC parlance, *qrels*). A relevance assessment ra is a tuple whose attributes include a query id, a document id (together forming the primary key) and a relevance label. A finite set of relevance assessments is denoted as $RA = \{ra_1, ra_2, \ldots\}$. Finally, given a list of queries Q, we denote with $R = [r_1, r_2, \ldots]$ the list of *ranked results* for the queries. Each retrieved result r associates a query $r.q$ and a document $r.d$ with a relevance score $r.s$ that defines the ranking order. The primary key of a retrieved results list is the pair $(q.id, d.id)$. Moreover, a retrieved results list may exhibit additional metadata such as the query-document *features* commonly exploited in LTR scenarios [14].

3.2 IR Transformers

We exploit the IR data model discussed above to express retrieval operations in an IR system as transformations between queries and document. To this end, we leverage *function objects*. A function object is a construct allowing an object to be invoked or called as if it were an ordinary function. A function object has properties depending on the function object's specific implementation. These properties allow the explicit declaration of the configuration parameters of each instantiated function object.

We build our declarative IR language on a generic function object we call a *transformer*. In general, a transformer $f : Q \times R \mapsto Q \times R$ takes as input a list of queries Q and a list of retrieved documents R, and returns another list of queries Q' and another list of documents R'. Depending on the specific implementation of a transformer, both inputs and output can be partially specified, i.e., input can be just Q or R and/or output can be just Q' or R'. An optional input can be omitted, i.e., the input is assumed to be an empty list, or, if present, it is ignored by the transformer. An optional output is ignored by the transformer, and it is a verbatim copy of the corresponding input. As we will see, this definition of transformer is general enough to allow the composition of transformers.

In the following, we describe some classes of transformers to show how commonly deployed operations in IR experiments can be implemented as transformers.

Basic retrieval. A classical search operation can be expressed as a function Retrieve : $Q \mapsto R$:

$$R' = \text{Retrieve}().\text{transform}(Q). \tag{1}$$

In doing so, the Retrieve function transforms a list of queries Q in a list of retrieved results R, i.e., a set of documents D for each query q in Q. We may instantiate the Retrieve operation in many ways, for instance using different weighting models (selecting e.g., BM25, TF.IDF, language modelling and their parameters as properties of the transformer) or even using different retrieval systems (Indri, Terrier, Anserini, etc.).

$$R' = \text{Retrieve}(\text{"BM25"}).\text{transform}(Q). \tag{2}$$

Moreover, we assume that transform() is the default method in our transformer objects, and hence need not be specified, i.e., Equation (2) is equivalent to:

$$R' = \text{Retrieve}(\text{"BM25"})(Q) \tag{3}$$

Note that, in the following, if no properties are specified, we suppress the empty () to improve readability.

Query Rewriting. In many cases, a query might be rewritten by the IR system before passing to the ranking component. For instance, the sequential dependence proximity model [22] adds operators such as the Indri #1 and #uw8 operators containing pairs and sequences of query terms, in order to boost the scores of documents where the query terms appear in close proximity. Similarly, Peng *et al.* [24] describe a query rewriting operation, known as context-sensitive stemming, where for some query terms, alternative inflections are added to the query. Rewriting can easily be expressed as a transformer like Rewrite : $Q \mapsto Q$, e.g.,:

$$Q' = \text{Rewrite}(Q). \tag{4}$$

Query Expansion. In pseudo-relevance feedback (PRF) query expansion, additional query terms can be added to the query based on how they occur in documents highly-ranked for the initial query. The identification of the refined query, can be expressed as a transformer, Expand : $Q \times R \mapsto Q$, e.g.:

$$Q' = \text{Expand}(Q, R). \tag{5}$$

The reweighted query should then be re-executed on the original index. Hence, the whole PRF process can be expressed as a combination of three transformers: a first Retrieve transformer,

a Expand transformer, which takes the queries Q and retrieved documents R, and calculates reformulated queries Q' by examination of the top-ranked documents for each query, and a second Retrieve transformer, to process the reformulated queries, e.g.,

$$R' = \text{Retrieve}(\text{Expand}(\text{Retrieve}(Q))). \tag{6}$$

Feature extraction. With the advent of LTR, multi-stage ranking pipelines have become commonplace in IR experiments. In the classical LTR paradigm, for a given query, k documents are ranked by an initial retrieval approach to form a *candidate set*; $k = 1000$ and BM25 form a typical setup [19]. Upon this candidate set, a number of additional retrieval features are extracted or calculated. For instance, PageRank or URL length are examples of *query independent* features that might be used in web search settings; proximity, field-based weighting models [20] are examples *query dependent* features. Both query-independent and query-dependent feature extraction can be expressed as transformers. Without loss of generality, we can use a single transformer $\text{Extract} : Q \times R \mapsto Q \times R$ to encompass all feature extraction processes, e.g.,

$$Q', R' = \text{Extract}(Q, R), \tag{7}$$

where Q is optional when extracting query-independent features.

Reranking. In multi-stage ranking pipelines, after feature extraction a candidate set of documents is re-ranked to boost effectiveness. Beside reranking using a LTR technique such as LambdaMART, more recently neural re-rankers such as BERT (e.g., [15]) are being increasingly widely used for improved effectiveness. In all cases, a re-ranker takes an input set of documents, and computes a new score for that set of documents, and hence a reranker can be expressed as the transformer $\text{Rerank} : Q \times R \mapsto R$, and a two-stage retrieval pipeline with feature extraction can be expressed as a combination of transformers, e.g.,

$$R' = \text{Rerank}(\text{Extract}(\text{Retrieve}(Q))). \tag{8}$$

Most re-rankers exploit machine learning techniques, hence they must be trained on some test data. To trigger the training of a re-ranker, this transformer exposes a method to estimate the model parameters to be used in subsequent IR experiments:

$$\text{Rerank.fit}(Q_{train}, RA_{train}, Q_{valid}, RA_{valid}), \tag{9}$$

where the parameters $Q_{train}, RA_{train}, Q_{valid}$, and RA_{valid} denote the training queries and qrels, and the validation queries and qrels, respectively, used to train the underlying machine-learned model.

Table 1 summarises the transformer classes presented. The optional input/output queries/retrieved documents are in parenthesis. Finally, we note that any arbitrary function that takes Q and/or R and returns Q and/or R can be used as a transformer, thereby allowing easy extensibility.

Table 1: Classes of Transformers. The optional input/output queries/retrieved documents are in parentheses.

Input	Output	Transformer
Q ($\times R$)	Q' ($\times R$)	Query rewriting
Q ($\times R$)	($Q \times$) R'	Basic Retrieval
$Q \times R$	Q' ($\times R$)	Query expansion
$Q \times R$	($Q \times$) R'	Re-ranking
$Q \times R$	$Q' \times R'$	Feature extraction

Table 2: PyTerrier operators for combining transformers.

Op.	Name	Description
>>	*then*	Pass the output from one transformer to the next transformer
+	*linear combine*	Sum the query-document scores of the two retrieved results lists
*	*scalar product*	Multiply the query-document scores of a retrieved results list by a scalar
**	*feature union*	Combine two retrieved results lists as features
|	*set union*	Make the set union of documents from the two retrieved results lists
&	*set intersection*	Make the set intersection of the two retrieved results lists
%	*rank cutoff*	Shorten a retrieved results list to the first K elements
^	*concatenate*	Add the retrieved results list from one transformer to the bottom of the other

3.3 Operators for Transformers

The notation for nested transformers calls in Equation (8) is difficult to write and hides the fact that a first-stage retrieval occurs before a second-stage re-ranking. To make it easy to combine different transformers in a succinct and easily understandable manner, we are inspired by deep learning frameworks towards creating succinct *pipelines* of IR transformations by *operator overloading*. In this way, we can use Python-like operators to allow simple notations for retrieval pipelines. In the following, we leverage some notations from relational algebra to describe these operators, as follows:

- Let $R_1 \bowtie R_2$ denote the natural join between two retrieved results lists. The result of the natural join is the set of all combinations of retrieved results in R_1 and R_2 that are equal on both of their composite $(q.id, d.id)$ primary key attributes;
- Let ${}_a\Gamma_b(R)$ denote the sorting of tuples in R according to the ascending values of attribute b after grouping by attribute a.
- Let ${}_a\sigma_K(R)$ denote the selection of the first K tuples in R after grouping by attribute a.
- Let $R[f(a_1, ..) \rightarrow b]$ denote the transformation of one or more attributes $a_1, ..$ of the tuples in R to a new attribute b according to function $f(\cdot)$.
- Following [26, p.236], let ${}_a\mathcal{G}_{\text{op}(b)}(R)$ denote the application of operator $\text{op}(\cdot)$ to attribute b of the tuples in R after grouping by attribute a, i.e., equivalent to a SQL statement projecting an aggregate function on b after a GROUP BY on a.

We now describe the transformer operators we have defined. A summary is provided in Table 2.

Linear combination (+). This operator allows two retrieval sets to be linearly combined, to support CombSUM data fusion or linear interpolation of relevance scores. If T_1 and T_2 represent transformers returning the same set of queries Q as in input, then

$$\begin{aligned} Q, R' = (T_1 + T_2)(Q, R) :=& \\ R_1 = T_1(Q, R), \quad & R_2 = T_2(Q, R) \\ R' = (R_1 \bowtie R_2)[s_1 + s_2 \rightarrow s]& \end{aligned}$$

Scalar product (*). This operator allows the scores of a retrieval set to be multiplied by a scalar value α:

$$Q', R' = (\alpha * T)(Q, R) :=$$
$$Q', R_1 = T(Q, R)$$
$$R' = R_1[\alpha s \rightarrow s]$$

This permits the weighting of systems within a linear combination.

Set union (|) and intersection (&). These operators allow two retrieval sets to be combined – for example, combining the documents obtained with and without PRF for use as a candidate set [5]. Due to their inherent set properties, the resulting document scores are not defined, e.g., the scores assume the special value $\perp$. Thus, these operators are intended for use with further (re-)ranking. More specifically, if T_1 and T_2 represent transformers returning the same set of queries Q as in input, then the set union would be defined as:

$$Q, R' = (T_1 \mid T_2)(Q, R) :=$$
$$R_1 = T_1(Q, R), \quad R_2 = T_2(Q, R)$$
$$R' = (R_1 \cup R_2)[\perp \rightarrow s]$$

Rank cutoff (%). This operator allows a ranking to be truncated at a given rank K. Given a retrieval results list produced by the transformer T, the result of the T `%` K operation is a retrieval result containing, for each query, the K documents returned by T with the highest scores:

$$Q, R' = (Q, R) \ \texttt{\%}\ K :=$$
$$R_1 =_{q.id} \Gamma_{-s}(R)$$
$$R' =_{q.id} \sigma_K(R_1).$$

Concatenate (^). This operator appends a second ranking after a first one for each query. This is useful in cases where, for instance, we may re-rank a few documents using an expensive approach, such as BERT, then append the remainder of the ranking from the baseline. Documents appearing in the first ranking for each query are removed from the second ranking. Documents from the second ranking have their scores adjusted such that the highest ranked remaining document from the second ranking has a score just less than the lowest ranked document from the first ranking. More formally:

$$Q, R' = (T_1 \ \hat{}\ T_2)(Q, R) :=$$
$$R_1 = T_1(Q, R), \quad R_2 = T_2(Q, R)$$
$$\underline{S} =_{q.id} \mathcal{G}_{\min(s)}(R_1), \quad \overline{S} =_{q.id} \mathcal{G}_{\max(s)}(R_2 - R_1)$$
$$R' = R_1 \cup \left((R_2 - R_1) \bowtie \underline{S} \bowtie \overline{S}\right)[r_2.s - \underline{s}.s + \overline{s}.s - \epsilon \rightarrow r_2.s]$$

where ϵ is a small constant, e.g., $\epsilon = 0.001$, used to represent the minimum score difference between documents coming from R_1 and R_2.

Feature union (**). This operator is intended to allow different retrieval systems to be composed as features for LTR. More specifically, if T_1 and T_2 represent transformers returning the same set of queries Q and documents R as is input, then

$$Q, R' = (T_1 \ **\ T_2)(Q, R) :=$$
$$R_1 = T_1(Q, R), \quad R_2 = T_2(Q, R)$$
$$R' = (R_1 \bowtie R_2)[[f_1, f_2] \rightarrow f]$$

The resulting retrieved result list combines, for each $(q.id, d.id)$ tuple, the features f_1 and f_2, e.g., the metadata, of the two input retrieved result lists into a new list of features f.

Composition (>>). This operator denotes that the output of an IR transformer should be used as input to another IR transformer. More specifically, if T_1 and T_2 represent transformers, then

$$Q', R' = (T_1 >> T_2)(Q, R) :=$$
$$Q_1, R_1 = T_1(Q, R)$$
$$Q', R' = T_2(Q_1, R_1).$$

For brevity, we also describe the composition operator as *then*. This operator allows to "chain" transformers and/or operator together, to define experimental *pipelines*. As an illustration, use of the composition operator, allows Equation (8) to be succinctly written as:

$$\mathsf{Pipe} = \mathsf{Retrieve} >> \mathsf{Rerank}$$
$$R' = \mathsf{Pipe}(Q)$$

Note that the type of Pipe is also a transformer, and hence all operators can be applied upon that transformer. Moreover, if at least one of the composed transformers expose a fit method as in Eq. (9), then the composed pipeline exposes a fit method as well. This method triggers the training of all the machine-learned Rerank transformers in the pipeline. Other transformers are applied as necessary, in order to make the appropriate transformation of the queries into the required inputs for the fit method.

3.4 Experiment Abstraction on Pipelines

Having defined the suite of transformers and operators available, we now turn to actually running IR retrieval experiments. In a procedural fashion, a retrieval pipeline can be evaluated in three steps:

- obtain the queries Q and corresponding relevance the assessments RA;
- transform those queries into results using the pipeline, let say $R = \mathsf{Pipe}(Q)$;
- apply an evaluation tool, such as the ubiquitous `trec_eval` tools – or its Python bindings `pytrec_eval` [28] – on RA and R, to obtain effectiveness measures such as MAP or NDCG.

Application of these three procedural steps might be considered laborious. Besides a sound knowledge of the specific IR system software to used, experiments are typically run using multiple scripting tools generating a lot of output files to be evaluated and compared to the outcomes of different experiments with similar but different implementations. Instead, we are inspired by the Cornac framework[2] for conducting comparative recommender system experiments. Cornac provides a succinct Experiment abstraction that allows many recommender systems to be evaluated using the same datasets for the same evaluation measures, while ensuring fair setup across matter such as cross-validation splits.

To this end, we define an Experiment function in PyTerrier, which can apply a list of retrieval pipelines upon a common set of queries, and evaluates the resulting result set from each pipeline to obtain a common set of evaluation measures. Our Experiment function builds upon the `pytrec_eval` [28] tool. It is also of note that Experiment acts as a trigger for the application of the pipelines upon a query set. The syntax for our proposed implementation of the Experiment abstraction is:

$$\mathsf{Experiment}([\mathsf{Pipe}_1, \mathsf{Pipe}_2], Q, RA, [\text{"map"}, \text{"ndcg"}]).$$

The output of an Experiment execution is a table comparing the specified retrieval pipelines side-by-side.

[2] https://cornac.preferred.ai/

Listing 1: Example experiment for the document ranking task of the TREC 2019 Deep Learning track.

```
first_pass = Retrieve(index, "BM25")                          # initial retrieval
# prf defines the candidate documents to re-rank using the additional features
prf = first_pass >> RM3(index) >> Retrieve(index, "BM25")
sdm = SequentialDependence >> Retrieve(index, "BM25")         # rewrites the query to use proximity operators
bert = CEDRPipeline("vanilla_bert")                           # applies a BERT re-ranker
ltr = xgBoostPipeline(xgBoost({"rank": "ndcg"}))              # defines and configures the LambdaMART LTR stage
# combine the full pipeline, using query expansion scores, proximity and BERT as features.
full_pipeline = prf >> (sdm **  bert) >> ltr
full_pipeline.fit(tr_topics, tr_qrels, va_topics, va_qrels) # train the pipeline
# evaluate the pipelines. Report MAP and NDCG@10
Experiment([first_pass, prf, full_pipeline], test_topics, test_qrels, metrics=["ndcg_cut_10", "map"])
```

While an Experiment does not currently handle the fitting (training) of the pipeline, it is easy to consider variants that automatically handle k-fold cross validation. A further variant might be the introduction of a grid search functionality to determine the best settings for different components in the pipeline. Due to the compositional nature of a retrieval pipeline, the grid search would be able to cache the outcomes of earlier stages in the pipeline, such that later retrieval components could be varied without re-execution of all pipeline stages.

Combined with the previously described transformers and operators, our proposed Experiment function allows the researcher to focus on *what* is being evaluated, i.e., the stages of the retrieval pipeline, rather than on the order of execution. Indeed, through the resulting domain-specific programming environment, researchers can design IR pipelines in terms of the concepts of the approach (combine these models to make a good candidate set; express these models as re-rankers; combine and learn a LTR model using these re-rankers as features). Each component can be simply implemented as a transformer operating on queries and documents.

In essence, we believe that the aforementioned types of transformers and operators allows to address a plethora of different IR operations. We define primitive search operations, such as search, rewrite, rerank, which can be easily implemented using standard search engine toolkits - indeed, we already have implementations for the Anserini and Terrier platforms. Moreover, instantiation of a learned model can be easily achieved by appending final transformers for learned methods for sci-kit Learn, (e.g., Random Forests) or xgBoost [6] (e.g., the LambdaMART LTR algorithm [3]). We have also implemented transformer objects for neural re-ranking implementations such as CEDR [15].

Listing 1 provides an example instantiation of a retrieval experiment, demonstrating how different retrieval transformers might be combined into a comprehensive, yet easily understandable, retrieval pipeline. This particular example, which might be applied to the TREC Deep Learning Track, demonstrates a composed pipeline involving pseudo-relevance feedback, BERT and LambdaMART.

However, the capabilities of different search engine toolkits differ, and hence there may be more efficient ways to instantiate the same pipeline. In the next section, we further elaborate on this topic, demonstrating how different toolkits can be used and optimised to efficiently implement the same pipeline. This allows the conceptual design of a retrieval pipeline – as expressed using transformers and operators – to diverge from its logical implementation, as executed upon the IR toolkits.

4 IMPLEMENTING TRANSFORMERS AND OPERATORS

Our domain-specific declarative environment for IR search experiments allows us to focus on the logical design of our experiments. The output of this design activity is a computational data-flow graph, with search operations as nodes, data dependencies between search operations as edges, where queries and documents are passed along edges. Some of these operations represent transformers for primitive search operations such as search, rerank, rewrite (as summarised in Table 1), while others represent operators to combine these transformers in different ways (as summarised in Table 2).

For the actual execution of IR retrieval experiments, the logical design of the experiments, i.e., the composition of pipelines, must be is *compiled* targeting a specific IR software platforms. Depending on the execution platforms, transformer and operators can be combined together to allow more efficient execution of the experiment. To implement the notions of pipeline compilation and optimisation, we use a pattern matching algorithm to identify patterns of pipeline expressions on a given *subject* pipeline, and apply *graph rewriting patterns* to create the optimised version of the subject pipeline for a given IR platform. Each pattern applies equivalence rules [26, p.583] leveraging the MatchPy pattern matching library [13], which takes into account the associativity of operators. Since, as mentioned in Section 3, we may instantiate transformers in many ways, for instance using different weighting models or other static properties, compiled pipelines can implement multiple transformers and/or operators, and can optimise their runtime execution depending on the IR software platform, as we illustrate in the following examples.

Dynamic pruning optimisations. By use of the rank cutoff operator (%), typically we are applying a rank cutoff to a list of retrieved results generated by Retrieve transformer as a separate operation:

$$\text{top10} = \text{Retrieve}(\mathit{index}, \text{"BM25"}) \ \% \ 10. \tag{10}$$

However, retrieval systems based on dynamic pruning techniques – such as MaxScore, WAND or BlockMaxWAND – can process queries and retrieve results faster when the number of documents to retrieve are reduced [27], as the higher scoring threshold means that more documents can be pruned, i.e., skipped over during retrieval. Hence the previous two-steps transformer can be automatically compiled as follows:

$$\text{top10} = \text{AnseriniRetrieve}(\mathit{index}, \text{"BM25"}, 10),$$

where AnseriniRetrieve is an Anserini-specific implementation of the Retrieve transformer, which uses Lucene's BlockMaxWAND-based search backend [11].

Learning to rank optimisations. To compute additional query-dependent features when re-ranking, the inverted index posting lists must be scanned until the requested docids are identified. This represents a large computational overhead even if skipping is used [27]. Indeed, consider the following retrieval pipeline, which computes two additional query dependent features (query likelihood and TF.IDF):

$$\begin{aligned} &\text{first\_pass} = \text{Retrieve}(\mathit{index}, \text{"BM25"}) \\ &\text{tfidf} = \text{Retrieve}(\mathit{index}, \text{"TFIDF"}) \\ &\text{ql} = \text{Retrieve}(\mathit{index}, \text{"QL"}) \\ &\text{pipeline} = \text{first\_pass} >> (\text{tfidf} ** \text{ql}) \end{aligned} \tag{11}$$

As written, both additional query dependent retrieval features would result in additional access to the inverted index posting lists for each query.

Instead, there have been two search architectures for computing additional query dependent features proposed in the literature: (1) the *doc vectors* approach [1], where the direct index (which records the terms occurring in each document) is used for computing additional features, and (2) the *fat postings* approach [20], where the postings for the query terms of the documents that enter the final retrieved set are cached in main memory, allowing the fast computation of additional query-dependent features for the documents in the retrieved set. In both cases, when executing feature retrieval pipelines involving multiple query-dependent features, instead of computing the features one by one with multiple passes of the doc vectors/fat postings, the pipeline can be rewritten to a single Terrier retrieval operation that extracts the fat postings first, then computes all other features on the fat postings, without executing two expensive retrieval transformers:

$$\text{pipeline} = \text{FeatureRetrieve}(\mathit{index}, \text{"BM25"}, [\text{"TFIDF"}, \text{"QL"}]).$$

In the next section, we demonstrate the efficiency benefits of using an optimised pipeline upon retrieval operations involving two underlying IR platforms.

5 EXPERIMENTS

The aim of the following experiments is to show that the optimisation of retrieval pipelines, as proposed in Section 4, results in more efficient search executions. These experiments demonstrate that we can implement equivalent retrieval pipelines using multiple retrieval backends (Anserini and Terrier), which can be automatically optimised in different ways. In particular, we address the following research questions:

RQ 1. Does optimising the execution of rank cutoffs for dynamic pruning enhance the efficiency of an Anserini retrieval?

RQ 2. Does optimising the execution of LTR to use the fat postings for computing multiple query dependent features enhance the efficiency of a Terrier retrieval?

5.1 Experimental Setup

We perform experiments on two standard TREC corpora, namely TREC Disks 4&5, and ClueWeb09. We index each corpus using both Anserini and Terrier, while recording position information but otherwise using their default settings. This results in two indices for each corpus, containing 528,155 and 50,220,423 documents, respectively, for Disks 4& 5, and ClueWeb09. For queries, we use corresponding TREC query sets: for Disks 4&5, 250 topics from the TREC Robust track '04, applying short (denoted T), medium (TD) and long (TDN) topic formulations; for ClueWeb09, we use 200 query-only topics from the TREC Web track 2009-2012.

Table 3: Mean response time (MRT, in milliseconds) of Terrier and Anserini for RQs 1 & 2. MRTs are shown before (denoted Orig.) and after optimisation by rewriting (denoted Opt.). Δ is the % improvement between original & optimised.

Formulation	Robust'04 T		Robust'04 TD		Robust'04 TDN		ClueWeb09	
	MRT	Δ	MRT	Δ	MRT	Δ	MRT	Δ
RQ1 - Rank Cutoff Optimisation								
Terrier	135.5	-	151.1	-	314.7	-	1040.4	-
Anserini Orig.	106.1	-	173.9	-	365.1	-	292.4	-
Anserini Opt.	**4.95**	-95%	**24.65**	-85.8%	**104.85**	-28.7%	**107.4**	-63%
RQ2 - Learning to Rank Optimisation								
Anserini	1336.1	-	1740.2	-	2625.4	-	3101.2	-
Terrier Orig.	501.4	-	626.7		1032.9	-	2047.9	-
Terrier Opt.	**33.8**	-93%	**116.1**	-81%	**350.8**	-66%	**1255.8**	-39%

Experiments are conducted on a Centos Linux 7.2 server with 96GB RAM and 12-core Intel E5-2609 CPUs. We use a single thread for all experiments, and report mean response time (MRT) in milliseconds. All experiments are conducted on our proposed framework PyTerrier.[3] PyTerrier is implemented in Python, and targets Anserini and Terrier backends, which are both written in Java – indeed, we make use of the Pyjnius library[4] that permits easy interactions between Python and Java code.

5.2 RQ1 Results

In this experiment, we compare the response times of Terrier and Anserini in implementing the retrieval pipeline contained in Eq. (10). In particular, as Anserini uses Lucene's search engine backend based on BlockMaxWAND, the pipeline can be optimised. We further measure the response times of Terrier – which does not deploy any dynamic pruning techniques – for comparison. The top half of Table 3 provides the mean response times for the Terrier pipeline, and the unoptimised and optimised Anserini pipelines, on both the TREC Robust '04 and ClueWeb09 test collections.

From the results, it is apparent that for short title-only (T) queries on the Robust '04 index, Anserini benefits from use of the BlockMaxWAND dynamic pruning technique, although the benefits are less apparent for longer queries (TD and TDN), where Terrier outperforms Anserini. This is expected, as document-at-a-time dynamic pruning techniques such as BlockMaxWAND are known to be less efficient for longer queries [27].

Next, comparing Anserini Optimised and the Anserini Original retrieval pipelines, we see that by informing the Anserini backend of the number of documents required is 10 rather than 1000, mean response times are improved by up to 95% on Robust '04 (short queries) and 63% on ClueWeb09. Therefore, we conclude that, in answer to RQ1, applying the rank cutoff optimisation within the framework can result in marked efficiency benefits for researchers.

5.3 RQ2 Results

Next, we experiment to evaluate the efficiency of Anserini and Terrier in executing the retrieval pipeline contained in Eq. (11), i.e., retrieving a candidate set of documents for each query using BM25, before calculating additional two query dependent features, namely

[3] The source code for our experiments is at https://github.com/cmacdonald/pyterrier_ictir2020. [4] https://github.com/kivy/pyjnius

TF.IDF and query likelihood with Dirichlet smoothing. Further, recall that such a retrieval pipeline can be optimised for Terrier, using the fat framework [20] for calculating multiple query dependent features. The bottom half of Table 3 reports the resulting response times for the Anserini execution as well as the un-optimised and optimised Terrier executions.

From the table, we note that the Terrier implementation is faster at executing this complex retrieval pipeline than Anserini. This is expected – at the time of writing, Anserini's reranking implementation uses one "query" to the underlying Lucene index for every document in the candidate set being re-scored. On the other hand, Terrier's original implementation is faster, but still requires one backend retrieval operation for each pipeline component. Finally, the optimised formulation only requires one backend fat retrieval operation for each query. In doing so, this formulation makes use of the fat framework to allow the efficient calculations of multiple query dependent features in a single pass of the query term's posting lists. Finally, we note that the benefit of the fat framework decreases as the length of the queries increase (T → TD → TDN); this implies that the overheads of keeping the postings around for lots of query terms causes additional memory pressures. The alternative doc vectors approach [1] may be more efficient in such situations. Therefore, in answer to RQ2, we find that automatic optimisation of retrieval pipelines for LTR have the potential to markedly enhance the efficiency of such experiments.

Finally, it is worth emphasising that the point of this experiment is not to demonstrate a "bake-off", but instead to show that a single retrieval pipeline – expressed in a conceptual manner – can be executed on PyTerrier using multiple different retrieval backends. Moreover, that pipeline can be executed by those backends in different manners, with different efficiencies. The researcher need not be knowledgeable about the capabilities of those backends.

6 CONCLUSIONS AND OUTLOOK

In this paper, we proposed a data model and framework for conducting IR experiments in a declarative manner. Our framework includes transformers representing standard retrieval operations, as well as operators for combining those transformers into retrieval pipelines. Further, we show how these pipelines can be automatically compiled and optimised, encoding knowledge of the capabilities of the underlying information retrieval system, to benefit the efficiency compared to semantically equivalent pipelines.

We believe that use of our framework can allow researchers to focus on creating transformers, for integrating their techniques with existing IR platforms – such as Anserini or Terrier – in end-to-end evaluation. The resulting code can be easily distributed as Jupyter notebooks, enhancing IR experiment reproducibility. In future, we believe that the proposed framework can be easily extended to support automatic parallelisation, by application of the pipeline using separate threads for different queries, as well as support for incremental querying, which would allow a neural re-ranker such as BERT to start training on some batches of training queries while the IR platform is still retrieving for further batches, rather than the current sequential executing of the retrieval pipeline.

ACKNOWLEDGEMENTS

The authors would like to acknowledge Alex Tsolov who contributed to the initial implementation of PyTerrier, along with other colleagues and contributors whose insights and contributions steered the PyTerrier development. Nicola Tonellotto was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

REFERENCES

[1] Nima Asadi and Jimmy J. Lin. 2013. Document vector representations for feature extraction in multi-stage document ranking. *Inf. Retr.* 16, 6 (2013), 747–768.
[2] Chris Buckley. 1985. *Implementation of the SMART Information Retrieval System.* Technical Report. USA.
[3] Chris J.C. Burges. 2010. *From RankNet to LambdaRank to LambdaMART: An Overview.* Technical Report MSR-TR-2010-82.
[4] Marc-Allen Cartright, Samuel Huston, and Henry Feild. 2012. Galago: A modular distributed processing and retrieval system. In *SIGIR 2012 Workshop on Open Source Information Retrieval.* 25–31.
[5] Shubham Chatterjee and Laura Dietz. 2019. Why Does This Entity Matter? Support Passage Retrieval for Entity Retrieval. In *Proc. ICTIR.* 221–224.
[6] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proc. SIGKDD.* 785–794.
[7] Apache Consortium. 2017. Solr Guide: Learning To Rank.
[8] Leif Azzopardi et al. 2017. Lucene4IR: Developing Information Retrieval Evaluation Resources Using Lucene. *SIGIR Forum* 50, 2 (2017), 58–75.
[9] Martín Abadi et al. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. OSDI.* USENIX Association, Savannah, GA, 265–283.
[10] Nicola Ferro and Diane Kelly. 2018. SIGIR Initiative to Implement ACM Artifact Review and Badging. *SIGIR Forum* 52, 1 (2018), 4–10.
[11] Adrien Grand, Robert Muir, Jim Ferenczi, and Jimmy Lin. 2020. From MAXSCORE to Block-Max Wand: The Story of How Lucene Significantly Improved Query Evaluation Performance. In *Proc. ECIR*, Vol. 12036. 20–27.
[12] Chris Kamphuis and Arjen. P. de Vries. 2019. Reproducible IR needs an (IR) (Graph) Query Language. In *SIGIR 2019 Open-Source IR Replicability Challenge.* 17–20.
[13] Manuel Krebber and Henrik Barthels. 2018. MatchPy: Pattern Matching in Python. *Journal of Open Source Software* 3, 26 (2018), 2.
[14] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.
[15] Sean MacAvaney, Andrew Yates, Arman Cohan, and Nazli Goharian. 2019. CEDR: Contextualized Embeddings for Document Ranking. In *Proc. SIGIR.* 1101–1104.
[16] Craig Macdonald. 2018. Combining Terrier with Apache Spark to Create Agile Experimental Information Retrieval Pipelines. In *Proc. SIGIR.* 1309–1312.
[17] Craig Macdonald, Richard McCreadie, and Iadh Ounis. 2018. Agile Information Retrieval Experimentation with Terrier Notebooks. In *Proc. DESIRES.*
[18] Craig Macdonald, Richard McCreadie, Rodrygo Santos, and Iadh Ounis. 2012. From Puppy to Maturity: Experiences in Developing Terrier. In *SIGIR 2012 Workshop in Open Source Information Retrieval.*
[19] Craig Macdonald, Rodrygo Santos, and Iadh Ounis. 2012. The whens and hows of learning to rank for web search. *Information Retrieval* (2012), 1–45.
[20] Craig Macdonald, Rodrygo Santos, Iadh Ounis, and Ben He. 2013. About Learning Models with Multiple Query Dependent Features. *ACM Tr. Inf. Sys.* 31, 3 (2013).
[21] Antonio Mallia, Michal Siedlaczek, Joel Mackenzie, and Torsten Suel. 2019. PISA: Performant Indexes and Search for Academia. In *SIGIR 2019 Open-Source IR Replicability Challenge.* 50–56.
[22] Donald Metzler and W Bruce Croft. 2005. A Markov random field model for term dependencies. In *Proc. SIGIR.* 472–479.
[23] Hannes Mühleisen, Thaer Samar, Jimmy Lin, and Arjen de Vries. 2019. Old Dogs Are Great at New Tricks: Column Stores for IR Prototyping. In *Proc. SIGIR.* 863–866.
[24] Fuchun Peng, Nawaaz Ahmed, Xin Li, and Yumao Lu. 2007. Context Sensitive Stemming for Web Search. In *Proc. SIGIR.* 639–646.
[25] Tao Qin and Tie-Yan Liu. 2013. Introducing LETOR 4.0 Datasets. *CoRR* abs/1306.2597 (2013).
[26] Abraham Silberschatz, Henry Korth, and S. Sudarshan. 2009. *Database Systems Concepts* (6 ed.). McGraw-Hill, Inc.
[27] Nicola Tonellotto, Craig Macdonald, and Iadh Ounis. 2018. Efficient Query Processing for Scalable Web Search. *Foundations and Trends in Information Retrieval* 12, 4-5 (2018), 319–500.
[28] Christophe Van Gysel and Maarten de Rijke. 2018. Pytrec_eval: An Extremely Fast Python Interface to trec_eval. In *Proc. SIGIR.* 873–876.
[29] Peilin Yang, Hui Fang, and Jimmy Lin. 2017. Anserini: Enabling the Use of Lucene for Information Retrieval Research. In *Proc. SIGIR.* 1253–1256.
[30] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *Proc. HotCloud.*